Using Theorem Provers for Testing:

Foundations, Challenges and Future Directions

Burkhart Wolff

Université Paris-Sud, LRI, CNRS

Abstract

While Formal Testing and Theorem-Proving are still perceived as antagonisms by many, there is a growing research field using the combination of both to increase the applicability of Formal Methods in industry, in particular in the area of Safety-and Security critical systems requiring formal certifications.

In this talk, I will present research (partially funded by the Digiteo Foundation) around the HOL-TestGen System, which strives for a synthesis of interactive and automated theorem proving as well of different formal testing techniques. I will present results which are of mutual interest for both research areas as well as an outlook for future directions.

Overview

- Test vs. Proof: An old controversy
  - Can proofs guarantee the "Absence of Errors"
  - Are deductive verifiers "better" than testers?
  - Can we avoid Tests ? Or Reality ?

- HOL-TestGen: A verification and validation approach by Model-based Testing (MBT)

- HOL-TestGen: Achievements FOR Proofs

- The Future of (Model-based) Testing

Test vs. Proof:
An old controversy

- “Dijkstra's Verdict” :

  ' Program testing can be used to show the presence of bugs, but never to show their absence!

Test vs. Proof:
An old controversy

- "Dijkstra's Verdict" :

> , Program testing can be used to show the presence of bugs, but never to show their absence!

- Well, Dijkstra was party;
  so can he be trusted ?

Test vs. Proof:
An old controversy

- "Dijkstra's Verdict" :

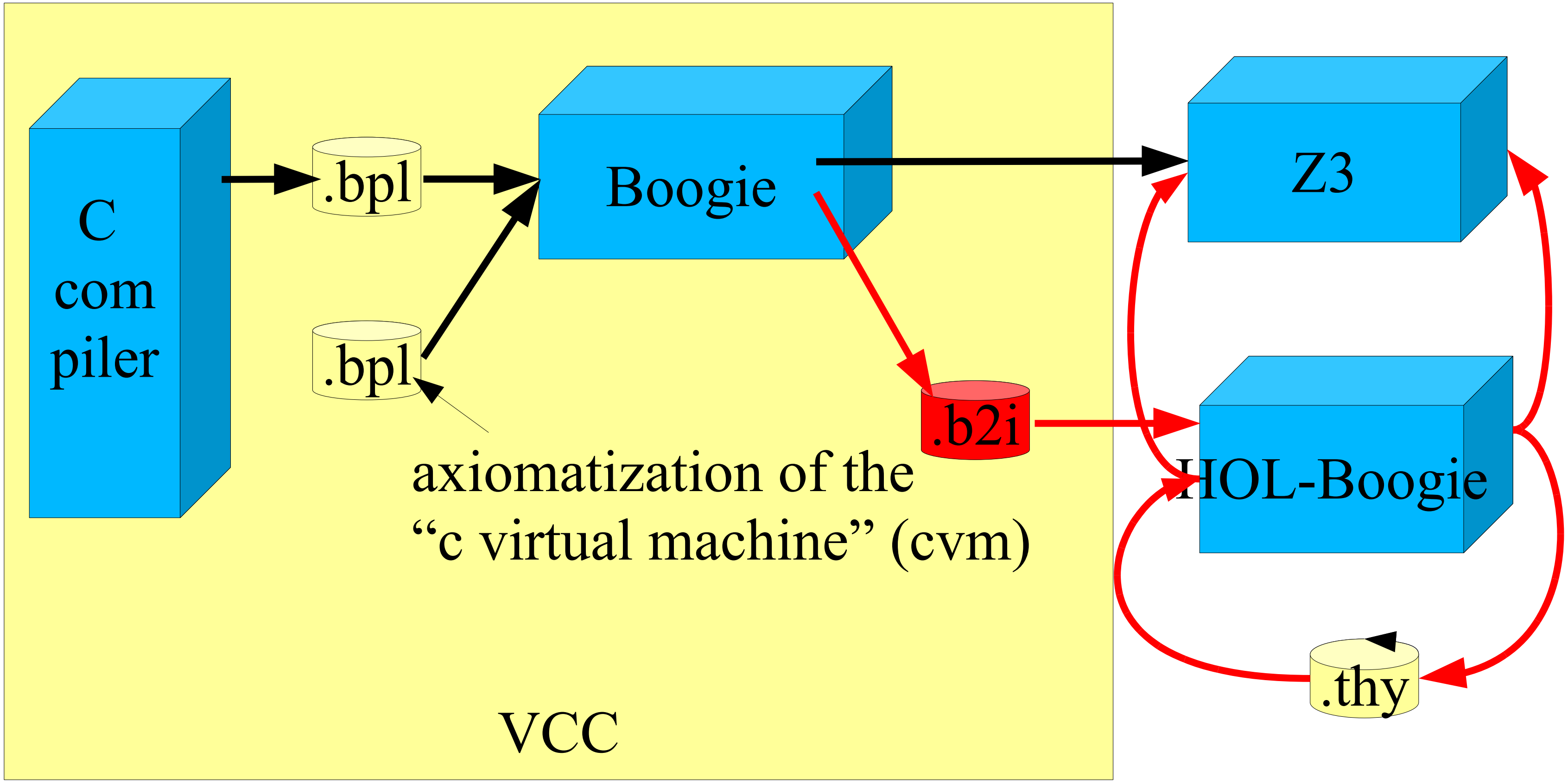
  ,   Program testing can be used to show the presence of bugs, but never to show their absence!

- So: can proof-based verifications guarantee the

  "abscence of bugs" ?

Test vs. Proof:
An old controversy

- An Architecture of a Program Verifier (VCC) HOL-Boogie [Böhme, Wolff]



Test vs. Proof: An old controversy

- The ugly reality:
  deductive verification methods
  make a lot of assumptions *besides being costly in brain-power!
    - operational semantics should be faithfully executed
    - complex memory-machine model
      consistent (VCC: 800 axioms)
    - correctness of the vc generation
      (for concurrent C with "ownership", "locks", … !):
    - correctness of the vc generator and prover
    - abscence of an environment (= Operating System)
      that manipulates the underlying state.

Test vs. Proof:
An old controversy

- Back to “Dijkstra's Verdict” :

  Program testing can be used to show the presence of bugs, but never to show their absence!

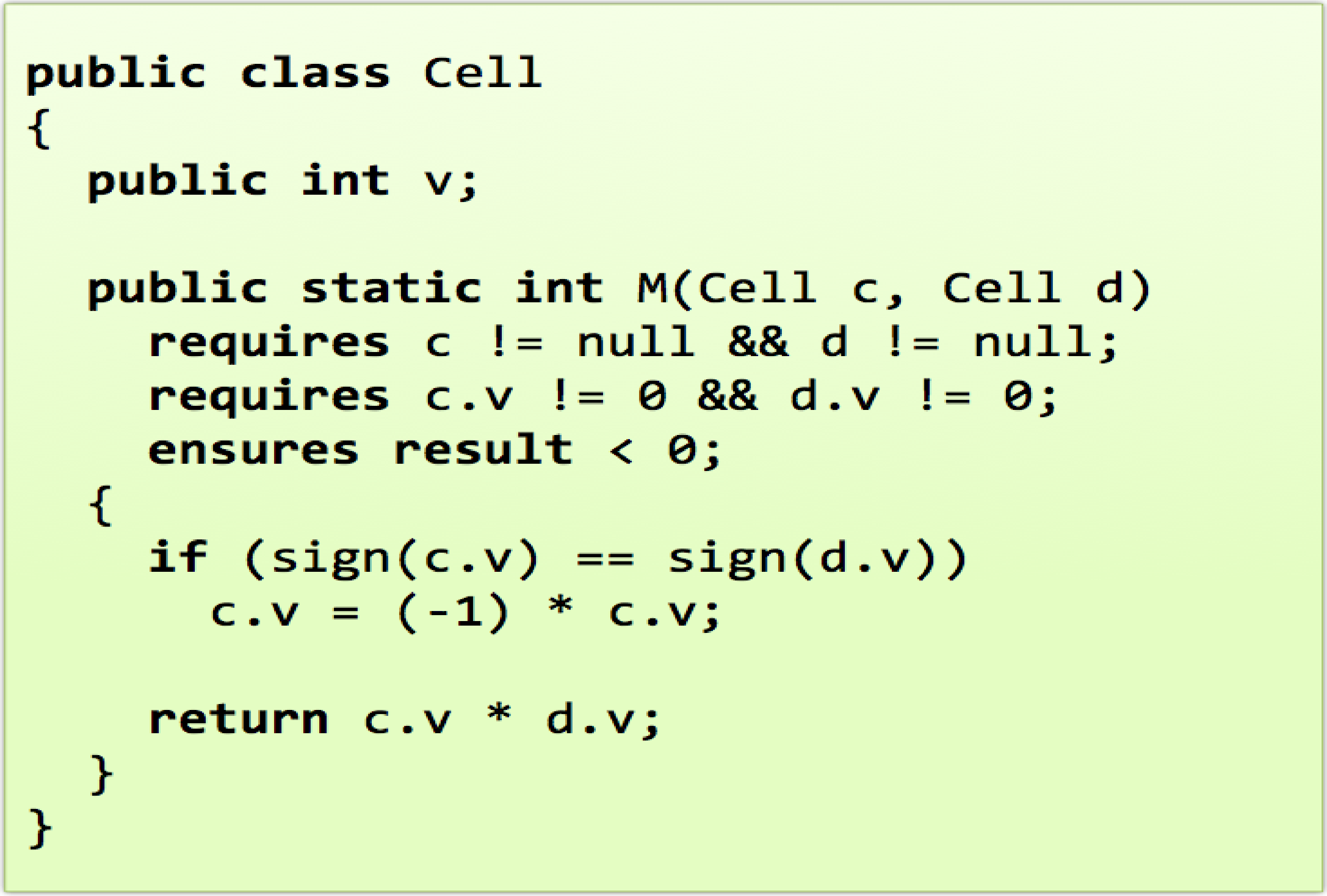
- Deductive Verification infers Properties on infinite sets of inputs; aren't they then

  “always better than tests” ?

Test vs. Proof:
An old controversy

- Well, this depends on these assumptions … See the (very nice) example of Maria Christakis,

  where for a simple program:



```
public class Cell
{
  public int v;

  public static int M(Cell c, Cell d)
    requires c != null && d != null;
    requires c.v != 0 && d.v != 0;
    ensures result < 0;
  {
    if (sign(c.v) == sign(d.v))
      c.v = (-1) * c.v;

    return c.v * d.v;
  }
}
```

Test vs. Proof: An old controversy

- Well, this depends on these assumptions …

  … two different tools
  - Clousot (deductive based verification)
  - Pex (white-box tester)

  provide alltogether differently false results, since their underlying assumptions on arithmetics and memory model are simply different.
  Accidently, the Pex-Verdict is actually more correct than Clousots …

Test vs. Proof: An old controversy

- "Dijkstra's Verdict" :

,   Program testing can be used to show the presence of bugs, but never to show their absence!

Can we actually always avoid testing ?

Test vs. Proof: An old controversy

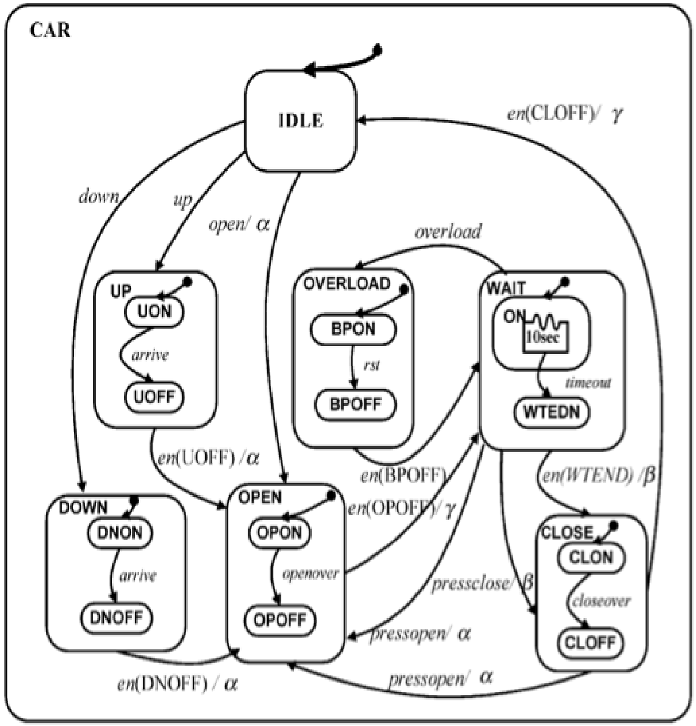
- “Dijkstra's Verdict” :

  Program testing can be used to show the presence of bugs, but never to show their absence!

- “Einsteins scepticism”:

  As far as the laws of mathematics refer to reality, they are not certain, as far as they are certain, they do not refer to reality.

Test vs. Proof:
An old controversy



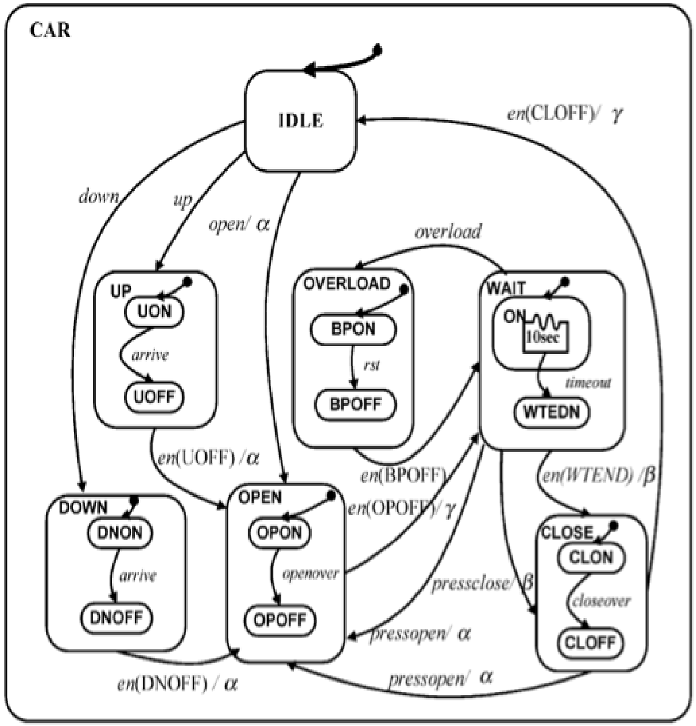
a posteriori

learning by experimenting

Model
(behaviour, and data !)

System
(hard + software)

Test vs. Proof:
An old controversy



a priori

test-case generation

a posteriori

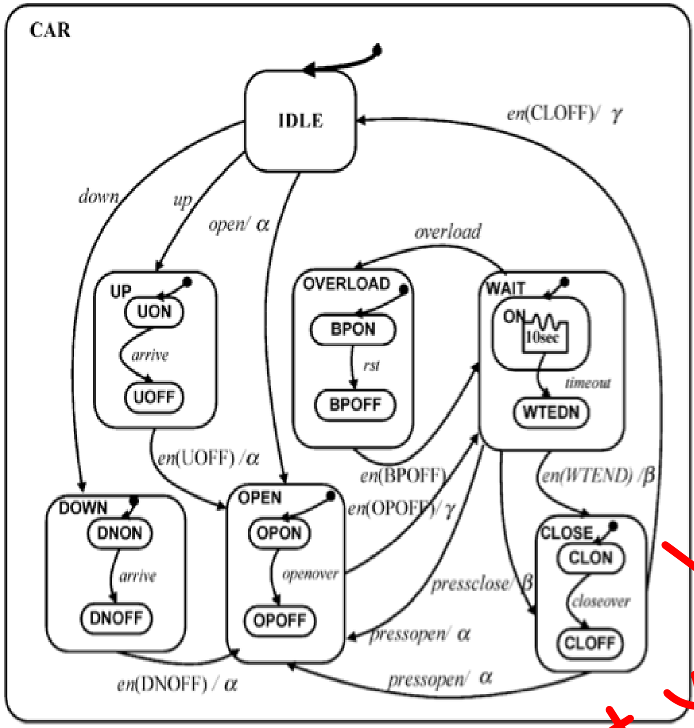
learning by experimenting

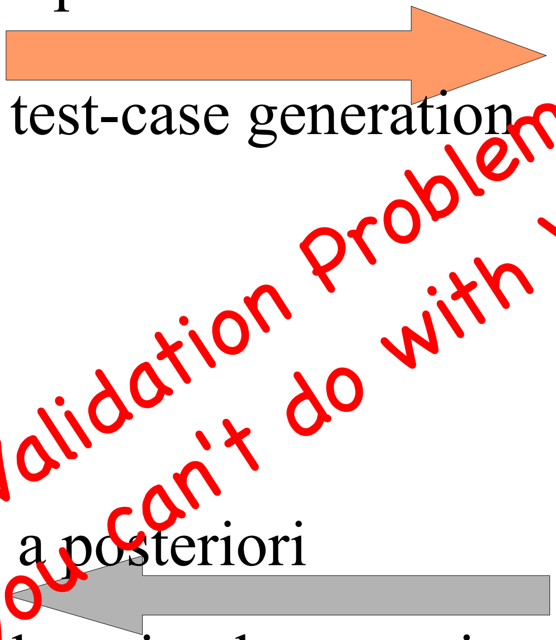
Model
(behaviour, and data !)

System
(hard + software)

Test vs. Proof:
An old controversy

.





a priori

test-case generation

a posteriori

learning by experimenting

Validation Problem:

What you can't do with Verification

Model
(behaviour, and data !)

System
(hard + software)

Verification by Model-based Testing …

- … can be done post-hoc; significant projects "reverse engineer" the model of a legacy system

- … attempts to find bugs in specifications EARLY (and can thus complement proof-based verification …)

- … can help system integration processes in a partly unknown environment ("embedded systems")

Nothing of this can be done by deductive verification methods !

Test vs. Proof:
Is it actually still a controversy?

- Dijkstra - Test :

  , Would Dijkstra fly with an aeroplane which is verified by deduct. methods alone ?

  ,

- Well, that's illegal.
  Certification bodies (CC, DO183) require tests,
  (and are very reluctant at proofs)

Test vs. Proof:
Is it actually still a controversy?

- Microsoft: Five major verification tools: Pex (Structural Test), SAGE(Fuzz Test) and Dafny, Spec#, VCC (VCG) use SMT solver Z3 !
- Test and Proofs, are they actually adversaries? (Tony Hoare, POPL2012, "says meanwhile no").

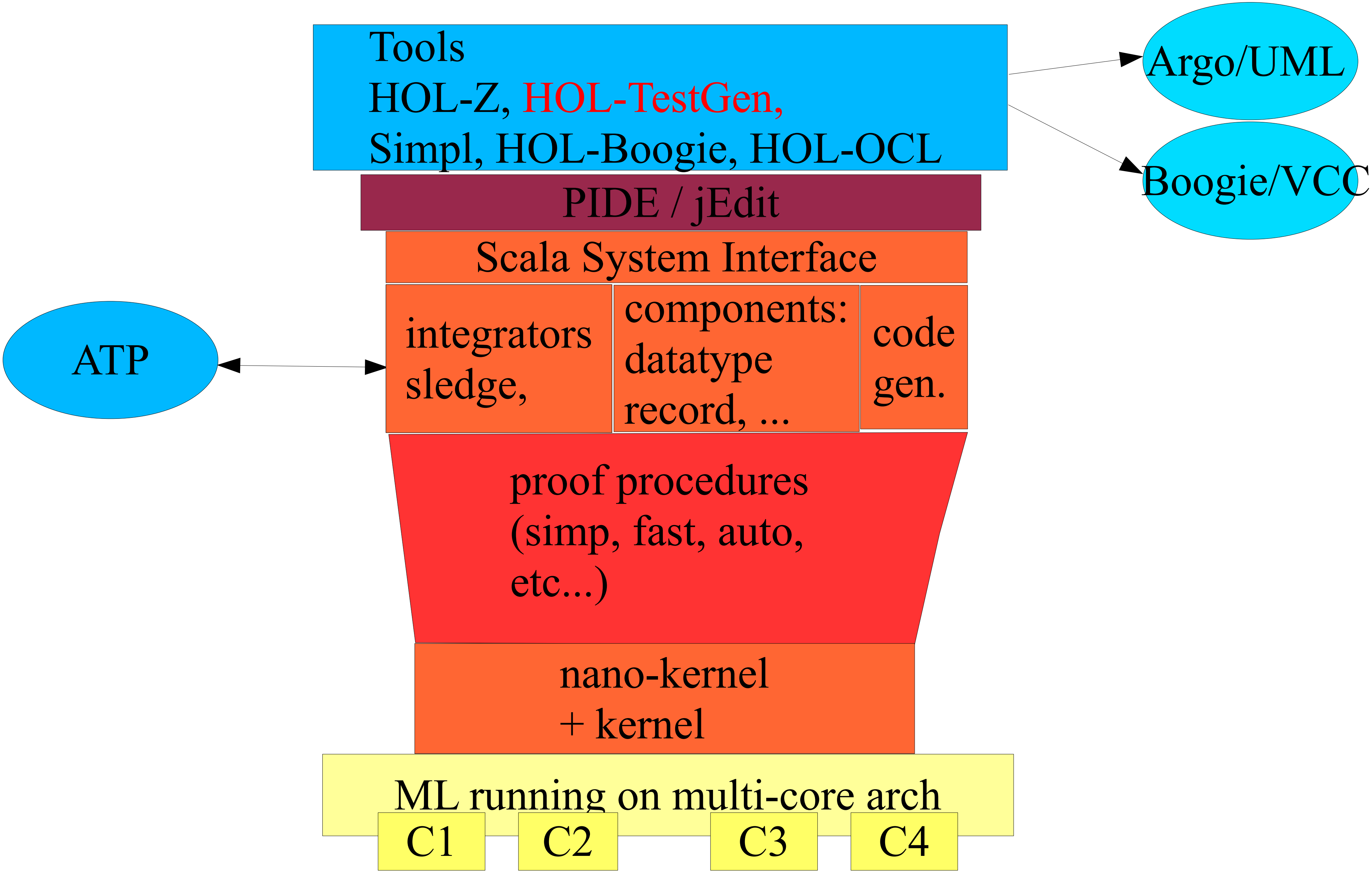
HOL-TestGen: A model-based approach to Verification

- Vision of HOL-Testgen
    - HOL-TestGen provides:
        - A formal testcase-generation method based on the solution of logical constraints

HOL-TestGen: A model-based approach to Verification

- HOL-TestGen provides:
    - A formal testcase-generation method based on the solution of logical constraints
    - Built-on top of an interactive theorem proving environment, it allows to combine automated provers with user intelligence

HOL-TestGen as Plugin in the Isabelle Architecture



Tools
HOL-Z, HOL-TestGen,
Simpl, HOL-Boogie, HOL-OCL

Argo/UML

Boogie/VCC

PIDE / jEdit

Scala System Interface

ATP

integrators
sledge,

components:
datatype
record, ...

code
gen.

proof procedures
(simp, fast, auto,
etc...)

nano-kernel
+ kernel

ML running on multi-core arch

C1 C2 C3 C4

Why Reusing Isabelle

Isabelle has:

… a lot of Infrastructure not worth to re-invent.

We us it as:

Formal Methods Tool Framework

"The ECLIPSE of FM - Tools"

HOL-TestGen:
“The Standard Workflow”

- Writing a test-theory (the “model”)

HOL-TestGen: “The Standard Workflow“

- Writing a test-theory (the “model”)

Example: Sorting in HOL

```
fun ins :: "(’a::linorder) ⇒’a list ⇒ ’a list"
where  "ins x [] = [x]"
      |"ins x (y#ys) = (if (x < y) then x#y#ys
                              else (y#(ins x ys)))"

fun sort:: "(’a::linorder) list ⇒ ’a list"
where   "sort [] = [] "
      | "sort (x#xs) = ins x (sort xs)"
```

HOL-TestGen:
"The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS

HOL-TestGen:
"The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS

pattern:

test_spec "pre $x \longrightarrow$ post x $(prog\, x)$"

HOL-TestGen:
“The Standard Workflow“

- Writing a test-theory
- Writing a test-specification TS

example:

```
test_spec "sort(l) = prog(l)"
```

HOL-TestGen:
“The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

(“Testcase Generation”)

HOL-TestGen:
“The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
  (“Testcase Generation”)

apply(gen_test_cases 3 1 “*prog*”)

HOL-TestGen:
“The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

  (“Testcase Generation”)

$$TC_1 \Rightarrow ... \Rightarrow TC_n \Rightarrow THYP(H_1) \Rightarrow \quad \Rightarrow THYP(H_m) \Rightarrow TS$$

  - where testcases TC_i have the form

    $$Constraint_1(x) \Rightarrow ... \Rightarrow Constraint_k(x) \Rightarrow P(\mathit{prog}\ x)$$

  - and where $THYP(H_i)$ are test-hypothesis

HOL-TestGen: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

Example:

$[] = prog([])$

$[?X1] = prog([?X1])$

$[?X1 \leq ?X2\,] \Longrightarrow [?X1, ?X2] = prog([?X1, ?X2])$

$[?X1 > ?X2\,] \Longrightarrow [?X2, ?X1] = prog([?X1, ?X2])$

HOL-TestGen:
“The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

...

5: $THYP(\exists x\ y.\ is\_sorted(PUT[x,y]) \longrightarrow$
$\forall x\ y.\ is\_sorted(PUT[x,y]))$

6: $is\_sorted(PUT\ [?X, ?Y, ?X])$

7: $THYP(\exists x\ y\ z.\ is\_sorted(PUT\ [x,y,z]) \longrightarrow$
$\forall x\ y\ z.\ is\_sorted(PUT\ [x,y,z]))$

8: $THYP(3 < |l| \longrightarrow is\_sorted(PUT\ l))$

HOL-TestGen: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

HOL-TestGen: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

`gen_test_data “...”`

HOL-TestGen:
"The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

[] = prog []

[3] = prog [3]

[6,8] = prog [6, 8]

[0,19] = prog [19, 0]

HOL-TestGen: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data
- Generating a test-harness

HOL-TestGen: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data
- Generating a test-harness
- Run of testharness and generation of test-document (a “test plan”)

HOL-TestGen:
A Larger Example: Red Black Trees

Red-Black-Trees: Test Specification

```
testspec :
(redinv t  ∧
 blackinv t)




     (redinv (delete x t)  ∧
      blackinv (delete x t))
```

where `delete` is the program under test.

Theory: Explicit Test-Hypothesises

- What to do with infinite data-strucutures ?
- What is the connection between test-cases and test statements and the test theorems?

⇒ Two problems, one answer:
Introducing Testhypothesis “on the fly” ...

$$\text{THYP} :: \text{“bool} \Rightarrow \text{bool”}$$
$$\text{THYP}(x) \equiv x$$

Theory of HOL-TestGen

- One type of test hypothesis:

  Uniformity-Hypothesis (for TestCase C)

  $$\mathrm{THYP}\ (\exists\ a \in C.\ P\ a\ \rightarrow \forall\ a \in C.\ P\ a)$$

Theory of HOL-TestGen

- Another: Regularity Hypothesis (τ, k)

- Consider the case $\tau = \text{list}(\alpha)$, $k = 2,3,4$:

$$\text{size}(x::\tau)<2 = (x = []) \vee (\exists\, a.\ x = [a])$$

$$\text{size}(x::\tau)<3 = (x = []) \vee (\exists\, a.\ x = [a]) \vee (\exists\, a\ b.\ x = [a,b])$$

$$\begin{aligned}\text{size}(x::\tau)<4 = {} & (x = []) \vee (\exists\, a.\ x = [a]) \vee (\exists\, a\ b.\ x = [a,b]) \\ & \vee (\exists\, a\ b\ c.\ x = [a,b,c])\end{aligned}$$

Theory of HOL-TestGen

- . . . derive the rule ($\tau\tau$ = list($\alpha\alpha$), $k = 2$):

$$\dfrac{\begin{array}{cccc} [x=[\,]] & [x=[a]] & [x=[a,b]] & \\ .. & .. & .. & \\ P & \bigwedge a.\quad P & \bigwedge a\ b.\quad P & \mathrm{THYP}\ M \end{array}}{P}$$

where $M = (\mathrm{size}\ x \geq > k \rightarrow P)$

- data separation lemma vs. “regularity hypothesis”

Theory Test-Hypothesis: Verifying Uniformity

- Reconsider the Uniformity Hypothesis:
  Case A: We test the hypothesis:

  5: $\mathrm{THYP}(\exists\, x\ y.\ y < x \rightarrow [y,x] = \mathrm{sort}(\mathrm{PUT}\ [x,y]) \rightarrow$
  $\forall\, x\ y.\ y < x \rightarrow [y,x] = \mathrm{sort}(\mathrm{PUT}\ [x,y]))$

  i.e. we state the hypothesis as test-spec!

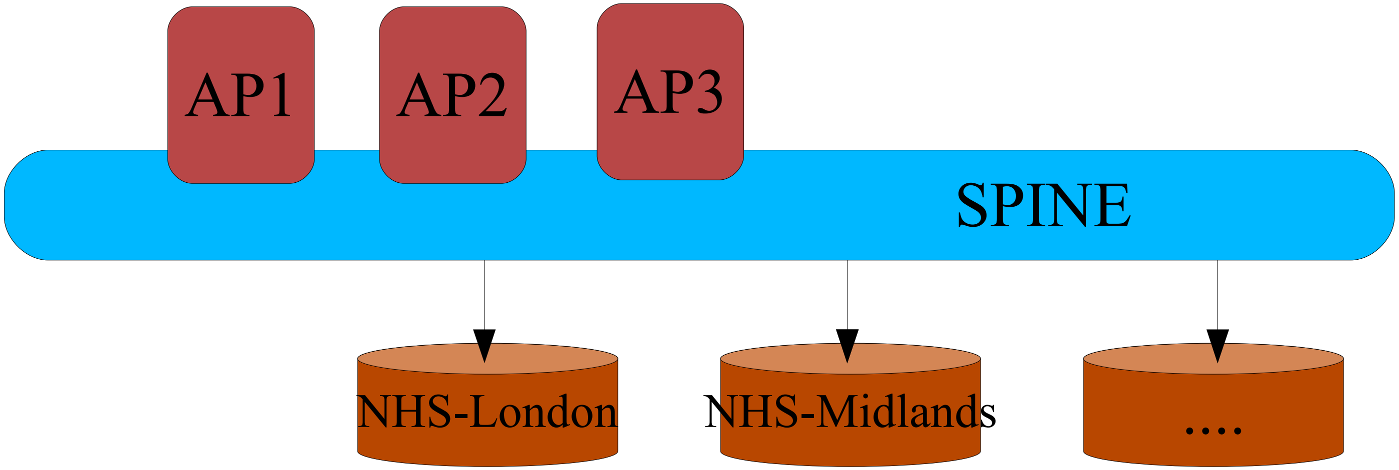
HOL-TestGen: Achievements FOR TP Community

- Larger Case-Studies in Test and Proof
  - NPfIT, Firewalls
  - Recently: Test-case generation for Hardware
  - EURO-MILS Projects in Certification CC
- Isabelle Distributions 2009-1, 2011, 2012

  including pervasive parallelisation of Kernel
- P-IDE Interface

HOL-TestGen:
Achievements FOR TP Community

- National Program for IT (NPfIT) :
  Large Case-Study together with British Telecom
- Test-Goal: NHS patient record access control mechanism
- Large Distributed, Heterogeneous System
- Legally required Access Control Policy
  (practically mostly enforced on the application level)



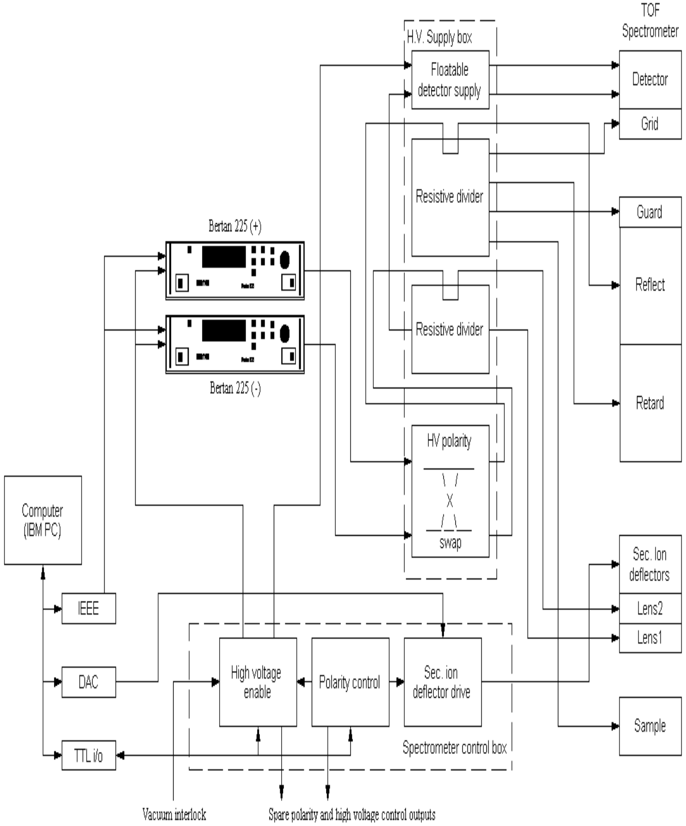
HOL-TestGen:
Achievements FOR TP Community

- National Program for IT (NPfIT) :
  Large Case-Study together with British Telecom
- Led to development of the

  Unified Policy Framework (Isabelle theory)
  encompassing RBAC, ARBAC and Firewall Policies

  Work on Verified Policy-Transformations
  led to interesting application in Network-
  Security testing

Models of Systems for Tests



a priori

test-case generation

a posteriori

learning by experimenting

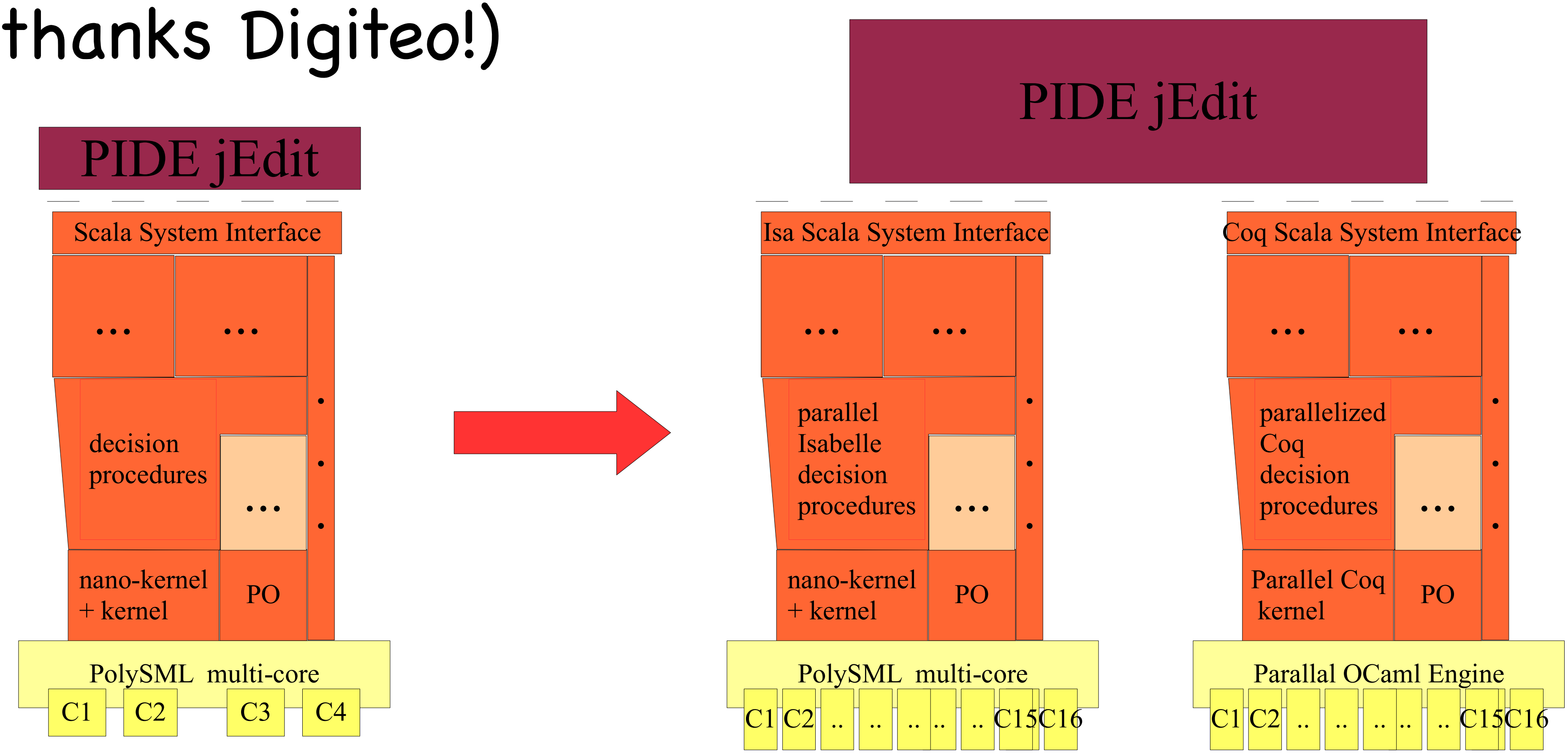
UPF Model instantiated with NPfIT AC Model

British Telecom Spine

HOL-TestGen: Achievements FOR TP Community

- ANR Project Paral ITP: Testing fueled the Parallelization in the Kernel of Isabelle

  (thanks Digiteo!)



PIDE jEdit

Scala System Interface

…

…

decision procedures

…

nano-kernel + kernel

PO

PolySML multi-core

C1 C2 C3 C4

PIDE jEdit

Isa Scala System Interface

…

…

parallel Isabelle decision procedures

…

nano-kernel + kernel

PO

PolySML multi-core

C1 C2 .. .. C15 C16

Coq Scala System Interface

…

…

parallelized Coq decision procedures

…

Parallel Coq kernel

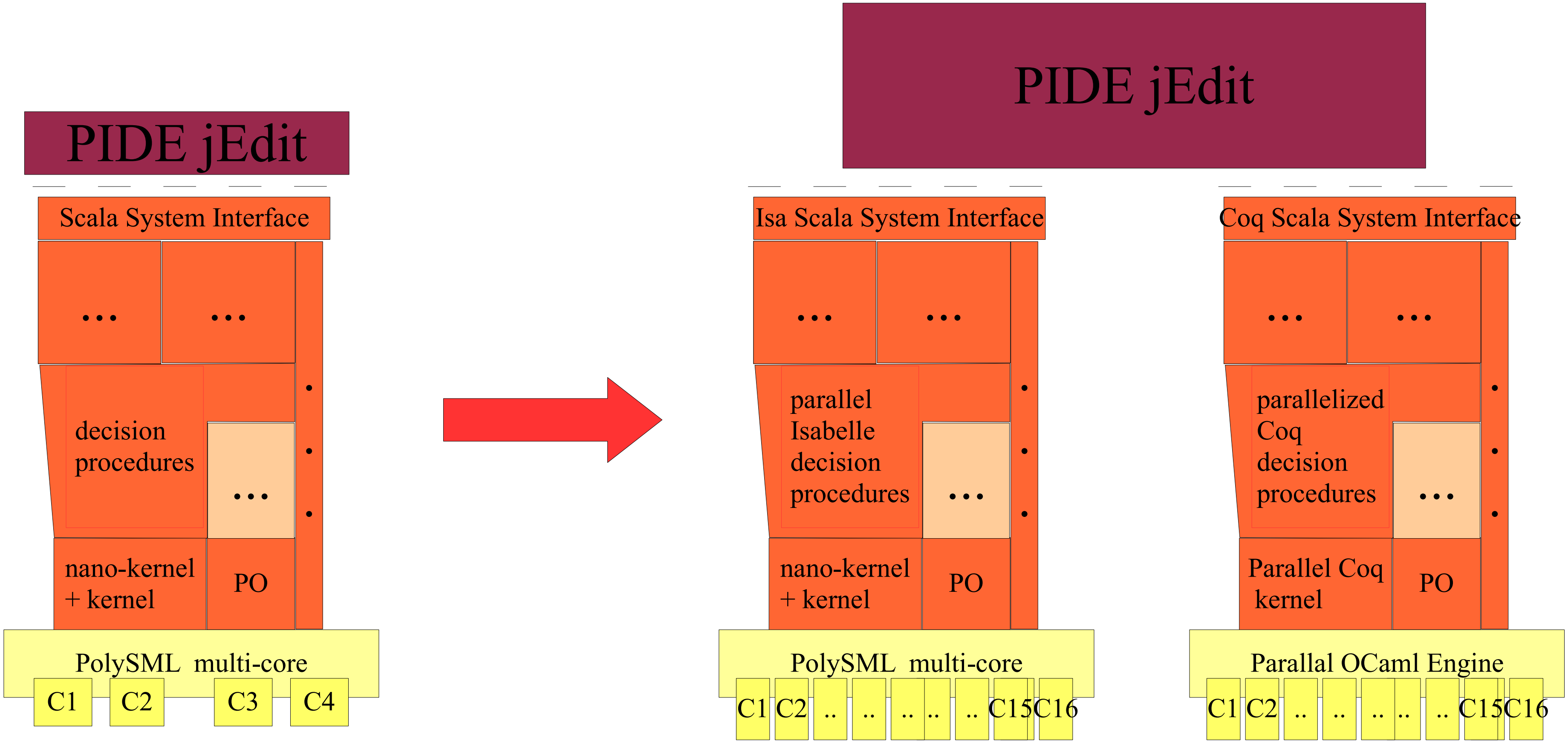
PO

Parallal OCaml Engine

C1 C2 .. .. C15 C16

HOL-TestGen: Achievements FOR TP Community

- . . . a technology which is now attempted to be transferred to Coq …



PIDE jEdit

Scala System Interface

…

…

decision procedures

…

nano-kernel + kernel

PO

PolySML multi-core

C1 C2 C3 C4

PIDE jEdit

Isa Scala System Interface

…

…

parallel Isabelle decision procedures

…

nano-kernel + kernel

PO

PolySML multi-core

C1 C2 .. .. C15 C16

Coq Scala System Interface

…

…

parallelized Coq decision procedures

…

Parallel Coq kernel

PO

Parallal OCaml Engine

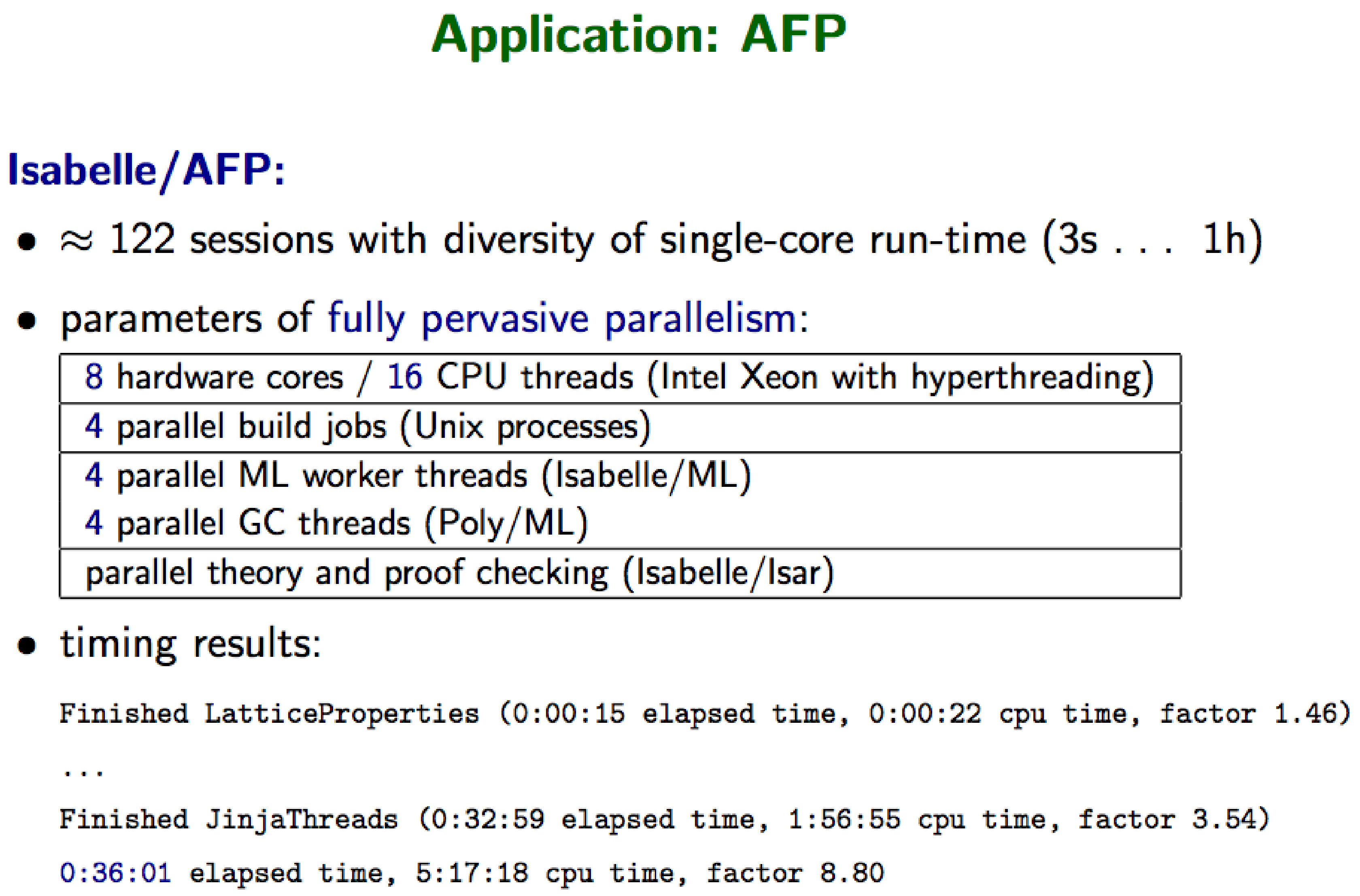
C1 C2 .. .. C15 C16

HOL-TestGen: Achievements FOR TP Community

- Isabelle: PIDE / jedit is meanwhile robust and stable and part of the Isabelle Distribution.

  Since Version 2013 the default interface.
- Support for advanced (nested) tool-tipping and hypertexting in the entire session.
- experiments with JAVA-Browsers.
- Coq: First Proof-of-Technologies to replace CoqIde available.

HOL-TestGen:
Achievements FOR TP Community



Application: AFP

Isabelle/AFP:

- ≈ 122 sessions with diversity of single-core run-time (3s . . . 1h)
- parameters of fully pervasive parallelism:

8 hardware cores / 16 CPU threads (Intel Xeon with hyperthreading)
4 parallel build jobs (Unix processes)
4 parallel ML worker threads (Isabelle/ML) 4 parallel GC threads (Poly/ML)
parallel theory and proof checking (Isabelle/Isar)

- timing results:

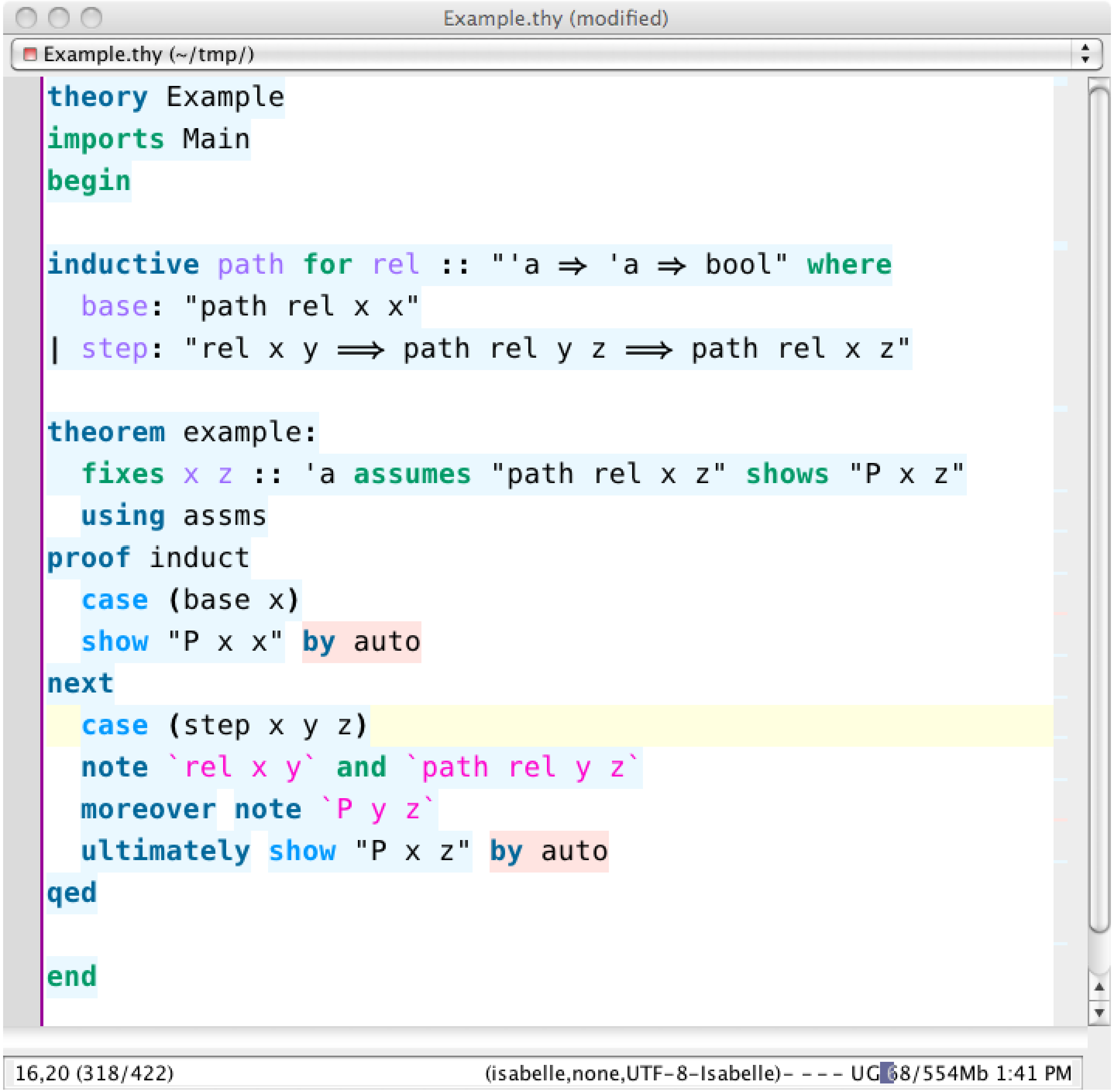
```
Finished LatticeProperties (0:00:15 elapsed time, 0:00:22 cpu time, factor 1.46)
...
Finished JinjaThreads (0:32:59 elapsed time, 1:56:55 cpu time, factor 3.54)
0:36:01 elapsed time, 5:17:18 cpu time, factor 8.80
```

Parallel fine-grained validation of structured proofs in

in the

jEdit - PIDE

(Isabelle2012-D)



```
theory Example
imports Main
begin

inductive path for rel :: "'a ⇒ 'a ⇒ bool" where
  base: "path rel x x"
| step: "rel x y ⟹ path rel y z ⟹ path rel x z"

theorem example:
  fixes x z :: 'a assumes "path rel x z" shows "P x z"
  using assms
proof induct
  case (base x)
  show "P x x" by auto
next
  case (step x y z)
  note `rel x y` and `path rel y z`
  moreover note `P y z`
  ultimately show "P x z" by auto
qed

end
```

```
16,20 (318/422)                (isabelle,none,UTF-8-Isabelle)- - - - UG 68/554Mb 1:41 PM
```

The Future of (Model-based) Testing

- More “Formal Methods under the Hood”
  - HOL-TestGenFW, SAGE, SAL,
    Excel-Expert, Crash-Analyser, …
- Cloudification
  - Excel-Expert …
- Gamification
  - Pex4Fun, Edutainment…
- Parallelization
  - In Test and Proof … but “no free lunch”…
- New Domain-Specializations
  - GUI-Testing, Model-Search, MKM …

Conclusion: Test & Proof

- … can never ever establish the absense of "Bugs" in a system! Never ever. Both of them.
- … can, when combined, further increase confidence in verification results by using mutually independent assumptions.
- … can, when combined, offer new ways to tackle abstraction and state space explosion. (Normalization Theorems, Massage of Constraint Systems, …)

Conclusion: HOL-Testgen ?

A formal testcase-generation method based on the solution of logical constraints

- Built-on top of an interactive theorem proving environment, it allows to combine automated provers with user intelligence
- has been applied in substantial case-studies (Firewall Case-Study, see TestCom/Fates 08)
- produces explicit test-hypothesis to establish a logical link between test and proof
- profits a lot from massive paralellization of symbolic computation ...

Sources Available

Version HOL-TestGen 1.7 (Isabelle 2011-1)

http://www.brucker.ch/projects/hol-testgen

Including Example Suite . . .

Case-Study: NPfIT

- Challenges:
  - access control rules for patient-identifiable information are complex and reflect the trade-off between patient confidentiality, usability, functional, and legislative constraints.
  - Traditional discretionary and mandatory access control and RBAC are insufficiently expressive to capture complex policies such as Legitimate Relationships, Sealed Envelopes or Patient Consent Management.
  - access rules of such a large system comprise not only elementary rules of data-access, but also access to security policies themselves enabling policy management. The latter is conventionally modeled in ABAC [6–8] and administrative RBAC [9, 10] models; A uniform modelling framework must be able to accommodate this.
  - The requirements are mandated by laws, official guidelines and ethical positions (e. g. [11, 12]) that are prone to change.

Case-Study: NPfIT

- Different "Information Gouvernance Principles" (= Policies):
  - Role-Based Access Control (RBAC): NPfIT uses administrative RBAC [9] to control who can access what system functionality. Each user is assigned one or more User Role Profile (URP). Each URP permits the user to perform several Activities.
  - Legitimate Relationship (LR): A user is only allowed to access the data of patients in whose care he is actually involved. Users are assigned to hierarchically ordered workgroups that reflect the organisational structure of a workplace.
  - Patient Consent (PC): Patients can opt out in having a Summary Care Record (SCR) at all, or to control uploads of data into the SCR. This requires additional mechanisms to manage consent.
  - Sealed Envelope (SE): The sealing concept is used to hide parts of an SCR from users. Kinds of seals: seal, seal and lock, clinician seal.

Modeling Framework:
Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - A Policy: A Decision Function
    (Modeling a “Policy Enforcement Point” in a System)

```
datatype α decision = allow α | deny α

types (α,β) policy = α ⇀ β decision    (* = α ⇒ β option *)

notation  α ↦ β = (α,β) policy
```

Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Policy Constructors

definition $\emptyset \equiv \lambda\, y.\ \text{None}$ (* $\emptyset :: \alpha \mapsto \beta$ *)

definition $p(x +\!\!\mapsto t) \equiv p(x \mapsto \text{Some}(\text{allow } t))$ (* $p :: \alpha \mapsto \beta$ *)
$p(x -\!\!\mapsto t) \equiv p(x \mapsto \text{Some}(\text{deny } t))$ (* where $p(x \mapsto t) \equiv$
$\lambda\, y.\ \text{if } y = x \text{ then } A \text{ else } p\ y$ *)

definition (*AllowAll :: "$(\alpha \rightharpoonup \beta) \Rightarrow (\alpha \mapsto \beta)$" *)

$\forall_A\, x.\ pf(x) \equiv (\lambda\, x.\ \text{case } pf\ x \text{ of } \text{Some } y \Rightarrow \text{Some}(\text{allow}(y))$
$\mid \text{None} \Rightarrow \text{None})$

(*DenyAll :: "$(\alpha \rightharpoonup \beta) \Rightarrow (\alpha \mapsto \beta)$"*)
$\forall_D\, x.\ pf(x) \equiv (\lambda\, x.\ \text{case } pf\ x \text{ of } \text{Some } y \Rightarrow \text{Some}(\text{allow}(y))$
$\mid \text{None} \Rightarrow \text{None})$

Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Domain, Range and Restictions on Policies (Z-like)

```
definition A ≡ {x.∃y. x = allow y},  D ≡ {x.∃y. x = deny y}

definition dom:: α ⇀ β ⇒ α set
where      dom f ≡ {x. f x ≠ None}

definition ran:: α ⇀ β ⇒ β set   ...


definition _    _ :: α set ⇒ α ⇀ β ⇒  α ⇀ β
where      S    p ≡ (λ x. if x ∈ S then p x else none)   (* domain restriction *)

definition _    _ :: α ⇀ β ⇒ α set ⇒ α ⇀ β ...   (* range restriction *)

definition _ ⊕ _ :: α ⇀ β ⇒  α ⇀ β ⇒  α ⇀ β ...(* first fit override  *)
```

Example: Firewalls

- Firewall Policies in UPF
  - Data:

    ip-address = int × int × int × int
    ip-packet = ip-address × protocol × content × ip-address

  - Firewall - Policies:

    policy : ip-packet ↦ ip-packet

    … this covers also Network Adress Translations (NAT's)

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

```
definition  me-ftp :: ip-packet ↦ ip-packet
where       me-ftp ≡  ∅((192,22,14,76),ftp,d,(192,22,14,76)
                         +↦(192,22,14,76),ftp,d,(192,22,14,76))
```

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

definition me-ftp :: ip-packet ↦ ip-packet
where me-ftp ≡ $\emptyset$((192,22,14,76),ftp,d,(192,22,14,76)
+↦(192,22,14,76),ftp,d,(192,22,14,76))

  - Combined Policies:

definition me-none-else:: ip-packet ↦ ip-packet

where me-none-else ≡ me-ftp $\oplus \forall_D x.\ x$

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

    definition me-ftp :: ip-packet $\Rightarrow$ ip-packet
    where me-ftp $\equiv$ $\emptyset((192,22,14,76),ftp,d,(192,22,14,76)$
    $+\mapsto(192,22,14,76),ftp,d,(192,22,14,76))$

  - Combined Policies:

    definition me-none-else:: ip-packet $\Rightarrow$ ip-packet

    where me-none-else $\equiv$ me-ftp $\oplus \forall_D x.\ x$